

The DimPy physical quantity package for Python

David Bate

Summer 2008

Contents

1	Basic operations involving units	2
1.1	Terminology	2
1.2	Creating quantities and new units	2
1.3	Quantity methods	3
1.4	Quantity functions	3
1.5	Dimensions	3
2	Defining additional units	3
2.1	Defining a new unit	4
3	Non-standard units	4
3.1	Flydims	4
3.2	Flyquants	4
4	Matrices containing physical quantities	5
4.1	Creation	5
4.2	Methods	6
4.3	Functions	6
4.4	Arithmetic	7
4.5	Reading values	7
5	Requesting Conversions	8
5.1	Computation	8
5.2	Prefixes and Suffixes	9
5.3	History	9
5.4	Defining variables	10
5.5	The online converter	10
6	Further work	10
6.1	A new Quantity class	10
6.2	Sage	11
6.3	Additional QuantMatrix functions	11

This documentation is available online in pdf and html formats at the DimPy website.

1 Basic operations involving units

1.1 Terminology

There are three tiers of dimensional quantity in DimPy:

- A `Dimension` stores the exponent of each SI unit in a `Quantity` or `Unit`.
- A `Unit` contains a `Dimension`, a unit name (“meter”) and a unit symbol (“m”). Meter, mile, second are units.
- A `Quantity` contains a `Unit` and a scalar multiple. Variables such as `my_height` and `mass_of_moon` would be `Quantity` instances.

When importing the `dimpy` namespace, `numpy` is also imported under `dimpy.numpy`.

1.2 Creating quantities and new units

In general, users do not need to explicitly construct a `Dimension`, `Unit` or `Quantity`. Instead, new `Units` and `Quantities` can be constructed from existing ones (note that in DimPy, all units appear in lowercase):

```
>>> my_height = 1.8*meter
>>> mass_of_moon = 7.36e22*kilogram
```

It is also possible to define new units from existing ones:

```
>>> Nm = newton*meter
>>> Nm
m N
```

When new units are created from a product of existing units, the unit symbol is generated from the factors. So here, `Nm` now has the unit symbol “m N”. However, this does not always give the result we want:

```
>>> mile = dimensionless(1600)*meter
>>> mile
m
```

The automatically generated name is a concatenation of the defining symbols’ names, which in this case is incorrect (`mile` has the unit symbol “mi”), so we need to change it. The `dispname` attribute is a dictionary, where each key is the unit symbol and the value is the power to which it appears:

```
>>> mile.dispname.clear()
>>> mile.dispname['mi']=1
>>> mile
mi
>>> Nm.dispname
{'m': 1, 'N': 1}
```

Note: `mile` is already defined in DimPy, see `Quantity.all_unit_names` for a list containing the names of all predefined units (2.) Also, observe the use of the `dimensionless()` function to create a dimensionless unit with a scale factor. If instead we asked for

```
>>> mile = 1600*meter
```

`mile` would be of type `Quantity` and would not have the additional properties of a `Unit`.

1.3 Quantity methods

To view a quantity in different units, the `Quantity.in_unit` method is used, returning a string with the required value. Alternatively the `%` operator may be used, however `%` has a low precedence in Python and so the right hand side should appear in parentheses if it is the composition of variables.

```
>>> my_height = 1.8*meter
>>> my_height.in_unit(foot)
'5.90551181102 ft'
>>> my_height % inch
'70.8661417323 inch'
```

1.4 Quantity functions

DimPy has the following functions for use with `Quantities`:

- `is_scalar_type(obj)` returns a boolean value determining if the object `obj` is a number (i.e. without any dimensional quantities).
- `get_dimensions(obj)` returns a tuple containing the dimensions and *on-the-fly* dimensions (3) of `obj`.
- `have_same_dimensions(obj1,obj2)` returns a boolean value determining if `obj1` and `obj2` have the same dimensions.
- `is_dimensionless(obj)` returns a boolean value determining if `obj` is dimensionless.

1.5 Dimensions

It is not necessary to create a `Dimension` for general use, but it is sometimes required as the argument of a function. The easiest way to create a `Dimension` is via keywords. Each key should be either the SI unit symbol or the physical quantity it measures, the value should be the exponent to which it appears:

```
>>> Dimension(length=1, kg=1, time=-2)
m kg s^-2
```

2 Defining additional units

The `derived_units` module creates derived units such as `mile` and `foot` when the DimPy package is imported, it also creates physical constants such as `G` and `c`. `derived_units` also provides the `dimensionless` function which can be used to create scaled units (1.2)

2.1 Defining a new unit

To define an additional unit, add an entry to the list of the form:

```
[unit_name, scalar_multiple, defining_units, unit_symbol]
```

where each variable should be a string. This unit will be automatically created and added to `Quantity.all_unit_names` when `DimPy` is initialised. For example:

```
['mile', '1609.344', 'meter', 'mi']
```

will create a variable called `mile` equal to `1609.344*meter` and have a unit symbol `mi`.

3 Non-standard units

Not every quantity one can consider is the product of SI units. Suppose a building contractor must construct “three houses per week”, then we need a way to create a unit based upon a string and the ability for these to interact with numbers and quantities.

3.1 Flydims

To create a unit from a string the `Flydim` class is used:

```
>>> house = Flydim('house')
>>> flat = Flydim('flat')
>>> house*flat
house flat
>>> house/flat
house flat^-1
```

Note that to invert a `Flydim`, the `invert` method should be used, as `1/flydim_instance` will return a `Flyquant`.

```
>>> house.invert()
house^-1
```

The units in a `Flydim` are stored in a dictionary, so one can define an empty `Flydim` and modify its contents directly:

```
>>> c = Flydim()
>>> c.dim['garage']=2
>>> print c
garage^2
```

3.2 Flyquants

A `Flyquant` contains a `Quantity` and a `Flydim`. `Flyquants` are usually constructed from already existing `Flydims` and `Quantities`:

```

>>> house = Flydim('house')
>>> street = 200*house
>>> length_of_house = 10*meter
>>> length_of_street = street*(length_of_house/house)
>>> length_of_street
2000.0 m

```

Arithmetic operations will return the simplest type of object:

```

>>> B = Flyquant(Flydim(""),1*meter)
>>> C = Flyquant(Flydim(""),Quantity(4))
>>> type(2*meter+B)
<class 'units.Quantity'>
>>> type(C + 8)
<type 'float'>

```

The functions `is_scalar_type(obj)`, `get_dimensions(obj)`, `have_same_dimensions(obj1,obj2)`, `is_dimensionless(obj)` and the `in_unit` method are also defined for `Flyquants` (1.4). The `fly(obj)` function will cast a number, `Unit` or `Quantity` into a `Flyquant`.

4 Matrices containing physical quantities

If one were to populate a large `numpy.matrix` `A` with `Quantity` objects, operations performed on `A` would be computationally slow. A `QuantMatrix` is a `numpy.matrix` associated with two `Unit` vectors, where the dimension of an entry in the matrix is calculated from the outer product of the two vectors. One should view a `QuantMatrix` as follows:

```

      kg  mol
m 1.0  2.0
s 3.0  4.0

```

which represents the matrix:

```

1.0 m kg  2.0 m mol
3.0 s kg  4.0 s mol

```

4.1 Creation

To create a `QuantMatrix` call:

```
QuantMatrix(matrix, [vertical_unit_vector, horizontal_unit_vector])
```

The elements of the unit vectors may have type `Dimension`, `Unit` or `Quantity`. `DimPy` will then calibrate the base matrix so that the matrix is displayed in SI units (and only `Dimension` types are stored):

```

>>> base_matrix = numpy.array([[1,2],[3,4]])
>>> vertical = [meter, second]
>>> horizontal = [mile, mole]

```

```
>>> A = QuantMatrix(base_matrix, [vertical, horizontal])
>>> A
      m      mol
m 1609.344  2.0
s 4828.032  4.0
```

The base matrix or quantities can be changed after creation using the `raw_numbers` and `quantities` attributes, but DimPy will check that the new values are compatible (i.e. that the size of the new matrix matches that of the old one).

```
>>> A.raw_numbers = numpy.array([[3,3],[3,3]])
>>> A
      m      mol
m 3      3
s 3      3
>>> A.quantities = [[meter, meter],[second,second]]
>>> A
      s      s
m 3.0    3.0
m 3.0    3.0
>>> A.raw_numbers = numpy.array([[1,2,3],[4,5,6]])
Traceback (most recent call last):
dimpy.qmatrix.QuantMatrixError: Shape of given array, (2, 3), does not match
existing shape, (2, 2).
```

4.2 Methods

A `QuantMatrix` has methods `shape`, `trace`, `transpose` and `dtype`, these behave the same as in `numpy`.

4.3 Functions

- `qhomogeneous(size, dimension)` will create a unit vector of length `size` where each element has dimension `dimension`.
- `qmat(array)` will cast any two dimensional array into a `QuantMatrix` with dimensionless dimensions.
- `qcolumn_vector(raw_numbers, quantities)` will create a column vector from the list `raw_numbers` with dimensions from `quantities`.

```
>>> qcolumn_vector([1,2,3],[meter, second, mole])
      1
      m  1.0
      s  2.0
      mol 3.0
```

- `shuffle(qmatrix, shuffle_vector)` does not alter the value of a `QuantMatrix` but may be used to alter the appearance of a `QuantMatrix`. It multiplies each dimension in the horizontal dimensions by `shuffle_vector` and divides each vertical dimension by `shuffle_vector`:

```
>>> A
      m mol
m 1  2
s 3  4
>>> shuffle(A, meter/second); A
      m^2 s^-1 m s^-1 mol
      s      1      2
m^-1 s^2    3      4
```

`shuffle_vector` may be a `Dimension`, `Unit` or `Quantity`.

The functions `qidentity`, `qones` and `qzeros` behave the same as their `numpy` counterparts, returning a `QuantMatrix` with dimensionless units. They may also be given the `dtype` keyword argument — the associated matrix will then have entries of that form.

4.4 Arithmetic

Before adding or multiplying instances of `QuantMatrix`, `DimPy` will first check that the operations are valid using the following criterion.

Addition: Suppose we are computing $\mathbf{A}+\mathbf{B}$, let AL and AT be the left and top dimensions of \mathbf{A} respectively and similarly BL and BT for \mathbf{B} . The addition is valid if:

- The pointwise division AL/BL gives a list of dimensions which are all the same (i.e. AL/BL is homogeneous).
- Similarly for AT, BT .
- If C and D are these two common dimensions, $C*D$ must be dimensionless.

Multiplication: Suppose we are computing $\mathbf{A}*\mathbf{B}$. If exactly one of \mathbf{A} or \mathbf{B} is a scalar, we perform elementwise multiplication, otherwise we require:

- We need `A.shape()[1] == B.shape()[0]` (so that standard matrix multiplication is defined).
- We require the inner product of AT and BL to be defined (i.e. the sum is allowed, which occurs if each term has the same dimensions).

These raise a `QuantMatrixError` if illegal. To exponentiate a `QuantMatrix`, it must be legal to multiply that `QuantMatrix` by itself.

4.5 Reading values

To read values from a `QuantMatrix` users should use a single set of square brackets containing a single index or a tuple (as for `numpy.matrix`). The output is equivalent to that of `numpy.matrix`. This notation supports slicing:

```

>>> A
      m mol
m 1 2
s 3 4
>>> A[0,0]
1609.344 m^2
>>> A[0,:]
      m mol
m 1609.344 2.0
>>> A[:,0]
      m
m 1609.344
s 4828.032
>>> A[0]
      m mol
m 1609.344 2.0

```

5 Requesting Conversions

DimPy contains an infix parser which can also handle requests involving quantities. This can be accessed using the interactive session:

```
>>> python quantity_parser.py
```

or within the DimPy namespace using the `parse` function:

```
>>> dimpy.parse("3 meters")
3.0 m
```

5.1 Computation

Given an expression, DimPy will try to calculate its value and return an answer in SI units. A line is printed showing how the request was interpreted and the result:

```

---> 3 meters/(2 hours)*4 seconds
3*meter/(2*hour)*4*second = 0.001666666666667 m

```

The parser accepts two forms of multiplication and each will give a different interpretation. The `*` symbol behaves as the standard Python multiplication, so any expression appearing after it will begin on the numerator. Alternatively, a space may be inserted which will be interpreted as multiplication with much higher precedence. In general, the natural way to write the sentence dictates which should be used:

```

---> 1.0/ten million
1.0/(ten*million) = 1e-07
---> 1.0/ten*million
1.0/ten*million = 100000.0
---> 1/newton meter

```

```

1/(newton*meter) = 1.0 m^-2 kg^-1 s^2
---> 1/newton*meter
1/newton*meter = 1.0 kg^-1 s^2

```

5.2 Prefixes and Suffixes

A *word* is any string of non-whitespace characters, separated by whitespace. The parser will first of all consider a scalar multiple at the start of a word. It will then look for a SI prefix, followed by some quantity and then for a 's', to see if the word is plural. A number may then follow to represent an exponent. This exponent will act on the quantity and prefix, but not the scalar multiple. See `QuantityParser.SI_PREFIXES` and `QuantityParser.SI_PREFIXES_SHORT` for a full list of long and short prefixes and their values. A quantity may be any type defined in DimPy (including a defined variable, 5.4) except for a `QuantMatrix` or `Dimension`. Therefore, the most general word is of the form:

```

---> 1e3millimeters2
1*10^3*(0.001*meter)^2 = 0.001 m^2

```

However, all of these components are optional.

5.3 History

When in interactive mode, the `quantity_parser` module stores a history of recent queries. To view this history, enter

```
---> print_history()
```

at the prompt and `index:value` pairs are displayed. To recall a previous request enter `#` followed by the corresponding index anywhere in a request:

```

---> 3 meters in miles
3*meter = 0.00186411357671 * mile
---> print_history()
0: 3 meters in miles
---> #0*2
(3*meter)*2 = 6.0 m

```

To delete an entry from the history enter:

```
---> delete("entry index")
```

Be aware that after a history entry is deleted, the indices of all the entries after it are decreased by one. As recalling histories is dynamic (i.e. only the index to recall is stored) any references that exist in the history will point to different requests after the deletion.

```

---> 3 meters in miles
3*meter = 0.00186411357671 * mile
---> 2 seconds in hours
2*second = 0.0005555555555556 * hour

```

```

---> #0*2
(3*meter)*2 = 6.0 m
---> delete(0)
---> #1          (#1 is "#0*2" which now evaluates to (2*second)*2)
((2*second)*2) = 4.0 s

```

The up and down keys may also be used to recall previous inputs, as in the standard Python session.

5.4 Defining variables

Another way of storing values in an interactive `quantity_parser` session is to define variables. To do this simply write an expression of the form

```

---> new_variable = 3 meters
new_variable = 3*meter

```

The string on the left hand side must contain only letters and underscores, the right hand side must be any valid expression. To recall the value, use the variable name as for a regular variable.

```

---> new_variable = 3 meters
new_variable = 3*meter
---> new_variable*4
new_variable*4 = 12.0 m

```

One advantage of declaring variables over using the history is that variables are not dynamic. The value on the right hand side is evaluated when defining the variable and then stored, so the value of a variable will not change during the session (unless it is redefined).

5.5 The online converter

The online converter behaves as the offline parser, except that the history is displayed within the webpage and entries are deleted using the check boxes.

6 Further work

Here are some ideas I did not have time to implement during my project.

6.1 A new `Quantity` class

Towards the end of my project, I realised that there was a problem with the `Unit` module — I do not think that this module is particularly tidy (for example, there are unnecessary global variables defined) and I believe some of the classes could be implemented better.

- The `Unit` class should not be a child of `Quantity`. Instead `Quantity` should have `Unit` and `value` attributes and all of the `Unit` arithmetic should be performed inside the `Unit` class. The `Quantity` class would then remove any scalar multiple from the `Unit` attribute and multiply it to the `Quantity.value` attribute.

- This would also make it easier to create a new way of representing a `Quantity` or `Unit`. Currently, only the SI units in a `Quantity` are stored. Therefore when the `Quantity` is printed to the user, it is represented in SI units. This is unhelpful and a better system would, where possible, produce an answer in units as close to the defining ones as possible (of course, when we calculate `meter*mile` we would have to choose a particular unit to store).

6.2 Sage

DimPy can be loaded into Sage by placing the DimPy folder into

```
<sage-directory>/devel/sage/build/sage/
```

and issuing the command “`import sage.dimp`”. However, one can currently only use Sage as a notebook for DimPy: to store calculations in a file. It would be useful if some of the functionality of Sage could incorporate DimPy, such as allowing a `Quantity` to be substituted into a `SymbolicExpression` (I believe some coercion rules need to be defined) or to have the calculus functions treat dimensions correctly (i.e. allowing a distance to be differentiated with respect to a time and the units will behave correctly).

6.3 Additional `QuantMatrix` functions

The `QuantMatrix` class provides a base for any matrix containing dimensions that we may want to construct. It would be useful to have more functions which do common tasks involving such matrices, such as creating a derivative matrix, a particular term in a Taylor expansion or a change of basis matrix. Such matrices are examples of the `QuantMatrix` class (this is why a `QuantMatrix` has the form it does), however constructor functions are required for the class to be more friendly.